13 The MVC and Razor Pages filter pipeline

This chapter covers

- The filter pipeline and how it differs from middleware
- Creating custom filters to refactor complex action methods
- Using authorization filters to protect your action methods and Razor Pages
- Short-circuiting the filter pipeline to bypass action and page handler execution
- Injecting dependencies into filters

In part 1 I covered the MVC and Razor Pages frameworks of ASP.NET Core in some detail. You learned how routing is used to select an action method or Razor Page to execute. You also saw model binding, validation, and how to generate a response by returning an `IActionResult` from your actions and page handlers. In this chapter I'm going to head deeper into the MVC/Razor Pages frameworks and look at the *filter pipeline*, sometimes called the *action invocation pipeline*.

MVC and Razor Pages use several built-in filters to handle cross-cutting concerns, such as authorization (controlling which users can access which action methods and pages in your application). Any application that has the concept of users will use authorization filters as a minimum, but filters are much more powerful than this single use case.

This chapter describes the filter pipeline primarily in the context of an API controller request. You'll learn how to create custom filters that you can use in your own apps, and how you can use them to reduce duplicate code in your action methods. You'll learn how to customize your application's behavior for specific actions, as well as how to apply filters globally to modify all of the actions in your app.

You'll also learn how the filter pipeline applies to Razor Pages. The Razor Pages filter pipeline is almost identical to the MVC/API controller filter pipeline, so we'll focus on where it differs. You'll see how to use page filters in your Razor Pages and learn how they differ from action filters.

Think of the filter pipeline as a mini middleware pipeline running inside the MVC and Razor Pages frameworks. Like the middleware pipeline in ASP.NET Core, the filter pipeline consists of a series of components connected as a pipe, so the output of one filter feeds into the input of the next.

This chapter starts by looking at the similarities and differences between filters and middleware, and when you should choose one over the other. You'll learn about all the different types of filters and how they combine to create the filter pipeline for a request that reaches the MVC or Razor Pages framework.

In section 13.2 I'll take you through each filter type in detail, how they fit into the MVC pipeline, and what to use them for. For each one, I'll provide example implementations that you might use in your own application.

A key feature of filters is the ability to short-circuit a request by generating a response and halting progression through the filter pipeline. This is similar to the way short-circuiting works in middleware, but there are subtle differences. On top of that, the exact behavior is slightly different for each filter, and I cover that in section 13.3.

You typically add filters to the pipeline by implementing them as attributes added to your controller classes, action methods, and Razor Pages. Unfortunately, you can't easily use DI with attributes due to the limitations of C#. In section 13.4 I'll show you how to use the `ServiceFilterAttribute` and `TypeFilterAttribute` base classes to enable dependency injection in your filters.

Before we can start writing code, we should get to grips with the basics of the filter pipeline. The first section of this chapter explains what the pipeline is, why you might want to use it, and how it differs from the middleware pipeline.

13.1 Understanding filters and when to use them

In this section you'll learn all about the filter pipeline. You'll see where it fits in the lifecycle of a typical request, how it differs between MVC and Razor Pages, and how filters differ from middleware. You'll learn about the six types of filters, how you can add them to your own apps, and how to control the order in which they execute when handling a request.

The filter pipeline is a relatively simple concept, in that it provides *hooks* into the normal MVC request, as shown in figure 13.1. For example, say you wanted to ensure that users can create or edit products on an e-commerce app *only* if they're logged in. The app would redirect anonymous users to a login page instead of executing the action.

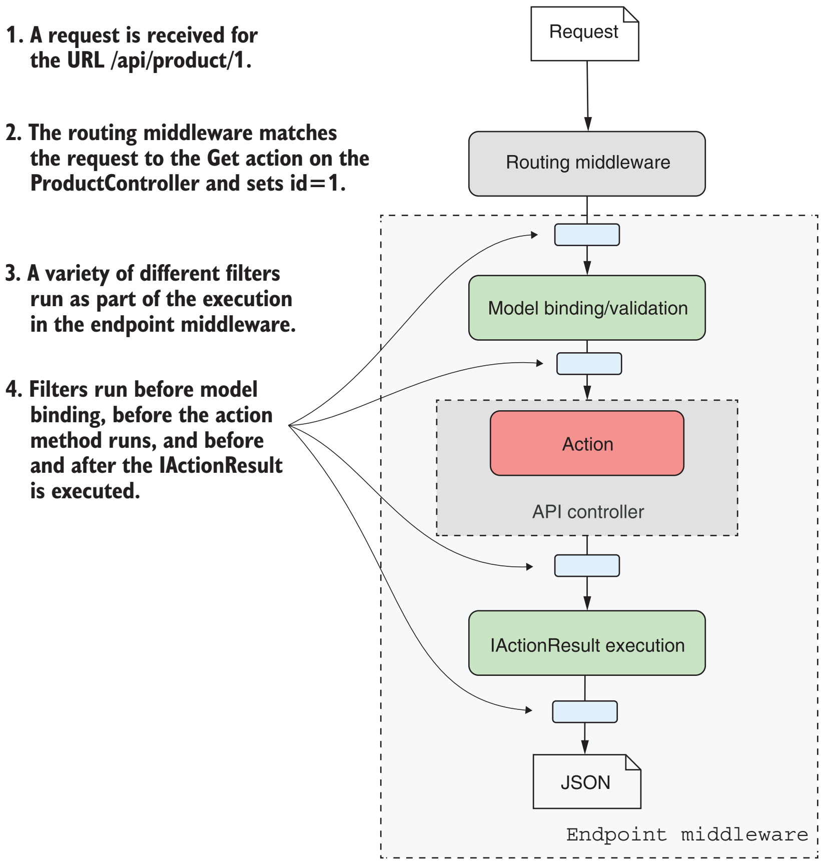

1. A request is received for the URL /api/product/1.
2. The routing middleware matches the request to the Get action on the ProductController and sets id=1.
3. A variety of different filters run as part of the execution in the endpoint middleware.
4. Filters run before model binding, before the action method runs, and before and after the IActionResult is executed.
Request
Routing middleware
Model binding/validation
Action
API controller
IActionResult execution
JSON
Endpoint middleware

Figure 13.1 Filters run at multiple points in the `EndpointMiddleware` as part of the normal handling of an MVC request. A similar pipeline exists for Razor Page requests.

Without filters, you'd need to include the same code to check for a logged-in user at the start of each specific action method. With this approach, the MVC framework would still execute the model binding and validation, even if the user were not logged in.

With filters, you can use the *hooks* in the MVC request to run common code across all, or a subset of, requests. This way you can do a wide range of things, such as

- Ensure a user is logged in before an action method, model binding, or validation runs
- Customize the output format of particular action methods
- Handle model validation failures before an action method is invoked
- Catch exceptions from an action method and handle them in a special way

In many ways, the filter pipeline is like a middleware pipeline, but restricted to MVC and Razor Pages requests only. Like middleware, filters are good for handling cross-cutting concerns for your application and are a useful tool for reducing code duplication in many cases.

13.1.1 The MVC filter pipeline

As you saw in figure 13.1, filters run at a number of different points in an MVC request. The linear view of an MVC request and the filter pipeline that I've used so far doesn't *quite* match up with how these filters execute. There are five types of filters that apply to MVC requests, each of which runs at a different *stage* in the MVC framework, as shown in figure 13.2.

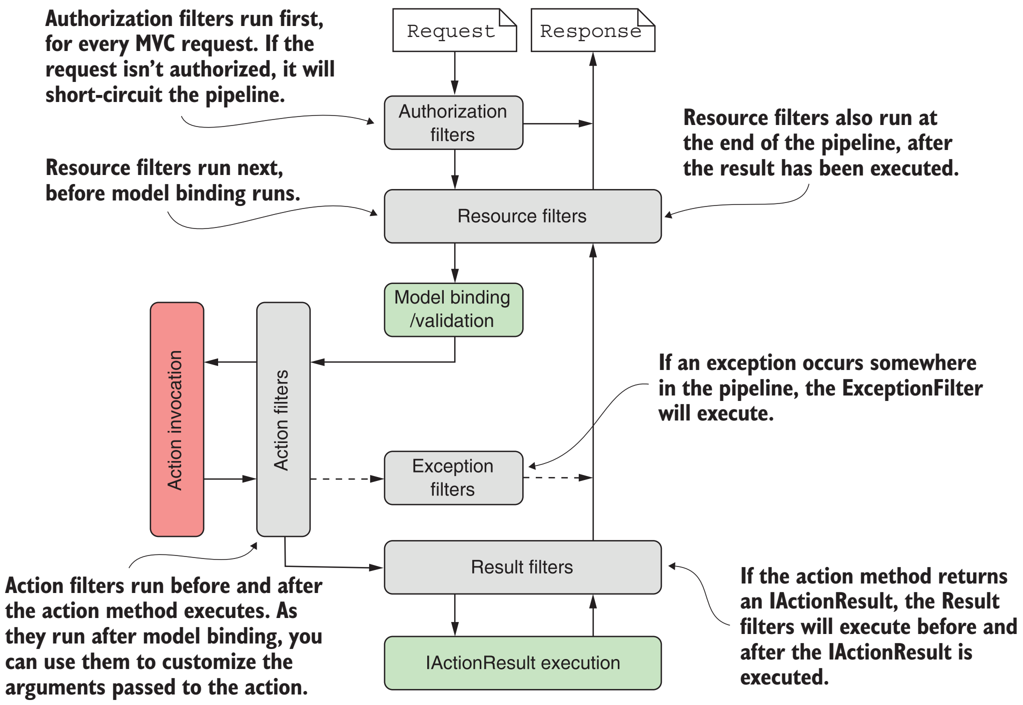

Authorization filters run first, for every MVC request. If the request isn't authorized, it will short-circuit the pipeline.
Request
Response
Authorization filters
Resource filters also run at the end of the pipeline, after the result has been executed.
Resource filters run next, before model binding runs.
Resource filters
Model binding /validation
Action invocation
Action filters
If an exception occurs somewhere in the pipeline, the ExceptionFilter will execute.
Exception filters
Result filters
Action filters run before and after the action method executes. As they run after model binding, you can use them to customize the arguments passed to the action.
If the action method returns an IActionResult, the Result filters will execute before and after the IActionResult is executed.
IActionResult execution

Figure 13.2 The MVC filter pipeline, including the five different filter stages. Some filter stages (resource, action, and result) run twice, before and after the remainder of the pipeline.

Each filter stage lends itself to a particular use case, thanks to its specific location in the pipeline, with respect to model binding, action execution, and result execution.

- *Authorization filters*—These run first in the pipeline, so they're useful for protecting your APIs and action methods. If an authorization filter deems the request unauthorized, it will short-circuit the request, preventing the rest of the filter pipeline (or action) from running.
- *Resource filters*—After authorization, resource filters are the next filters to run in the pipeline. They can also execute at the *end* of the pipeline, in much the same way that middleware components can handle both the incoming request and the outgoing response. Alternatively, resource filters can completely short-circuit the request pipeline and return a response directly.

  Thanks to their early position in the pipeline, resource filters can have a variety of uses. You could add metrics to an action method, prevent an action method from executing if an unsupported content type is requested, or, as they run before model binding, control the way model binding works for that request.
- *Action filters*—Action filters run just before and after an action method is executed. As model binding has already happened, action filters let you manipulate the arguments to the method—before it executes—or they can short-circuit the action completely and return a different `IActionResult`. Because they also run after the action executes, they can optionally customize an `IActionResult` returned by the action before the action result is executed.
- *Exception filters*—Exception filters can catch exceptions that occur in the filter pipeline and handle them appropriately. You can use exception filters to write custom MVC-specific error-handling code, which can be useful in some situations. For example, you could catch exceptions in API actions and format them differently from exceptions in your Razor Pages.
- *Result filters*—Result filters run before and after an action method's `IActionResult` is executed. You can use result filters to control the execution of the result, or even to short-circuit the execution of the result.

Exactly which filter you pick to implement will depend on the functionality you're trying to introduce. Want to short-circuit a request as early as possible? Resource filters are a good fit. Need access to the action method parameters? Use an action filter.

Think of the filter pipeline as a small middleware pipeline that lives by itself in the MVC framework. Alternatively, you could think of filters as *hooks* into the MVC action invocation process that let you run code at a particular point in a request's lifecycle.

This section described how the filter pipeline works for MVC controllers, such as you would use to create APIs; Razor Pages uses an almost identical filter pipeline.

13.1.2 The Razor Pages filter pipeline

The Razor Pages framework uses the same underlying architecture as API controllers, so it's perhaps not surprising that the filter pipeline is virtually identical. The only difference between the pipelines is that Razor Pages do not use action filters. Instead, they use page filters, as shown in figure 13.3.

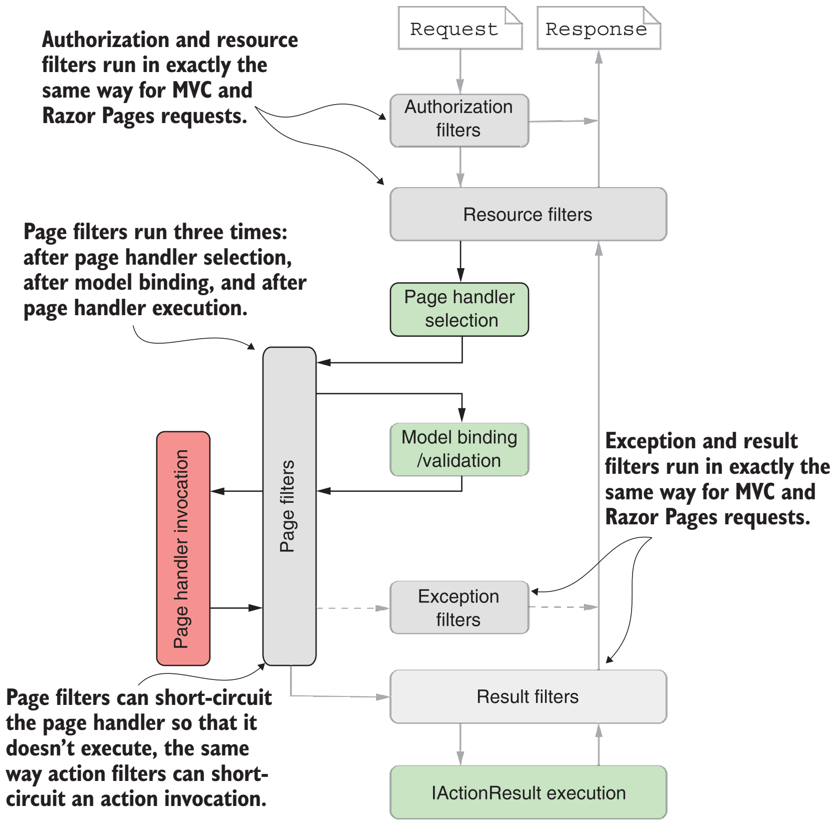

Request
Response
Authorization and resource filters run in exactly the same way for MVC and Razor Pages requests.
Authorization filters
Resource filters
Page filters run three times: after page handler selection, after model binding, and after page handler execution.
Page handler selection
Model binding /validation
Page handler invocation
Page filters
Exception and result filters run in exactly the same way for MVC and Razor Pages requests.
Exception filters
Result filters
Page filters can short-circuit the page handler so that it doesn't execute, the same way action filters can short-circuit an action invocation.
IActionResult execution

Figure 13.3 The Razor Pages filter pipeline, including the five different filter stages. Authorization, resource, exception, and result filters execute in exactly the same way as for the MVC pipeline. Page filters are specific to Razor Pages and execute in three places: after page hander selection, after model binding and validation, and after page handler execution.

The authorization, resource, exception, and result filters are exactly the same filters as you saw for the MVC pipeline. They execute in the same way, serve the same purposes, and can be short-circuited in the same way.

NOTE These filters are literally the same classes shared between the Razor Pages and MVC frameworks. For example, if you create an exception filter and register it globally, the filter will apply to all your API controllers and all your Razor Pages equally.

The difference with the Razor Pages filter pipeline is that it uses *page filters* instead of action filters. In contrast to other filter types, page filters run three times in the filter pipeline:

- *After page handler selection*—After the resource filters have executed, a page handler is selected, based on the request's HTTP verb and the `{handler}` route value, as you learned in chapter 5. After page handler selection, a page filter method executes for the first time. You can't short-circuit the pipeline at this stage, and model binding and validation has not yet executed.
- *After model binding*—After the first page filter execution, the request is model-bound to the Razor Page's binding models and is validated. This execution is highly analogous to the action filter execution for API controllers. At this point you could manipulate the model-bound data or short-circuit the page handler execution completely by returning a different `IActionResult`.
- *After page handler execution*—If you don't short-circuit the page handler execution, the page filter runs a third and final time after the page handler has executed. At this point you could customize the `IActionResult` returned by the page handler before the result is executed.

The triple execution of page filters makes it a bit harder to visualize the pipeline, but you can generally just think of them as beefed-up action filters. Everything you can do with an action filter, you can do with a page filter. Plus, you can hook in after page handler selection if necessary.

> **TIP** Each execution of a filter executes a different method of the appropriate interface, so it's easy to know where you are in the pipeline and to only execute a filter in one of its possible locations if you wish.

One of the main questions I hear when people learn about filters in ASP.NET Core is "Why do we need them?" If the filter pipeline is like a mini middleware pipeline, why not use a middleware component directly, instead of introducing the filter concept? That's an excellent point, which I'll tackle in the next section.

13.1.3 Filters or middleware: Which should you choose?

The filter pipeline is similar to the middleware pipeline in many ways, but there are several subtle differences that you should consider when deciding which approach to use. When considering the similarities, they have three main parallels:

- *Requests pass through a middleware component on the way "in" and responses pass through again on the way "out."* Resource, action, and result filters are also two-way, though authorization and exception filters run only once for a request, and page filters run three times.

- *Middleware can short-circuit a request by returning a response, instead of passing it on to later middleware.* Filters can also short-circuit the filter pipeline by returning a response.
- *Middleware is often used for cross-cutting application concerns, such as logging, performance profiling, and exception handling.* Filters also lend themselves to cross-cutting concerns.

In contrast, there are three main differences between middleware and filters:

- Middleware can run for all requests; filters will only run for requests that reach the `EndpointMiddleware` and execute an API controller action or Razor Page.
- Filters have access to MVC constructs such as `ModelState` and `IActionResults`. Middleware, in general, is independent from MVC and Razor Pages and works at a "lower level," so it can't use these concepts.
- Filters can be easily applied to a subset of requests; for example, all actions on a single controller, or a single Razor Page. Middleware doesn't have this concept as a first-class idea (though you could achieve something similar with custom middleware components).

That's all well and good, but how should we interpret these differences? When should we choose one over the other?

I like to think of middleware versus filters as a question of specificity. Middleware is the more general concept, which operates on lower-level primitives like the `HttpContext`, so it has the wider reach. If the functionality you need has no MVC-specific requirements, you should use a middleware component. Exception handling is a great example of this; exceptions could happen anywhere in your application, and you need to handle them, so using exception handling middleware makes sense.

On the other hand, if you *do* need access to MVC constructs, or you want to behave differently for some MVC actions, then you should consider using a filter. Ironically, this can also be applied to exception handling. You don't want exceptions in your Web API controllers to automatically generate HTML error pages when the client is expecting JSON. Instead, you could use an exception filter on your Web API actions to render the exception to JSON, while letting the exception handling middleware catch errors from Razor Pages in your app.

TIP Where possible, consider using middleware for cross-cutting concerns. Use filters when you need different behavior for different action methods, or where the functionality relies on MVC concepts like `ModelState` validation.

The middleware versus filters argument is a subtle one, and it doesn't matter which you choose as long as it works for you. You can even use middleware components *inside* the filter pipeline as filters, but that's outside the scope of this book.

TIP The middleware as filters feature was introduced in ASP.NET Core 1.1 and is also available in later versions. The canonical use case is for localizing requests to multiple languages. I have a blog series on how to use the feature here: http://mng.bz/RXa0.

Filters can be a little abstract in isolation, so in the next section we'll look at some code and learn how to write a custom filter in ASP.NET Core.

13.1.4 Creating a simple filter

In this section, I'll show you how to create your first filters; in section 13.1.5 you'll see how to apply them to MVC controllers and actions. We'll start small, creating filters that just write to the console, but in section 13.2 we'll look at some more practical examples and discuss some of their nuances.

You implement a filter for a given stage by implementing one of a pair of interfaces—one synchronous (sync), one asynchronous (async):

- *Authorization filters*—`IAuthorizationFilter` or `IAsyncAuthorizationFilter`
- *Resource filters*—`IResourceFilter` or `IAsyncResourceFilter`
- *Action filters*—`IActionFilter` or `IAsyncActionFilter`
- *Page filters*—`IPageFilter` or `IAsyncPageFilter`
- *Exception filters*—`IExceptionFilter` or `IAsyncExceptionFilter`
- *Result filters*—`IResultFilter` or `IAsyncResultFilter`

You can use any POCO class to implement a filter, but you'll typically implement them as C# attributes, which you can use to decorate your controllers, actions, and Razor Pages, as you'll see in section 13.1.5. You can achieve the same results with either the sync or async interface, so which you choose should depend on whether any services you call in the filter require async support.

NOTE You should implement *either* the sync interface *or* the async interface, *not both*. If you implement both, only the async interface will be used.

Listing 13.1 shows a resource filter that implements `IResourceFilter` and writes to the console when it executes. The `OnResourceExecuting` method is called when a request first reaches the resource filter stage of the filter pipeline. In contrast, the `OnResourceExecuted` method is called after the rest of the pipeline has executed: after model binding, action execution, result execution, and all intermediate filters have run.

Listing 13.1 Example resource filter implementing `IResourceFilter`

```
public class LogResourceFilter : Attribute, IResourceFilter
{
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        Console.WriteLine("Executing!");
    }
```

Executed at the start of the pipeline, after authorization filters

The context contains the HttpContext, routing details, and information about the current action.

```
    public void OnResourceExecuted(
        ResourceExecutedContext context)
    {
        Console.WriteLine("Executed"");
    }
}
```

Executed after model binding, action execution, and result execution

Contains additional context information, such as the IActionResult returned by the action

The interface methods are simple and are similar for each stage in the filter pipeline, passing a context object as a method parameter. Each of the two-method sync filters has an `*Executing` and an `*Executed` method. The type of the argument is different for each filter, but it contains all the details for the filter pipeline.

For example, the `ResourceExecutingContext` passed to the resource filter contains the `HttpContext` object itself, details about the route that selected this action, details about the action itself, and so on. Contexts for later filters will contain additional details, such as the action method arguments for an action filter and the `ModelState`.

The context object for the `ResourceExecutedContext` method is similar, but it also contains details about how the rest of the pipeline executed. You can check whether an unhandled exception occurred, you can see if another filter from the same stage short-circuited the pipeline, or you can see the `IActionResult` used to generate the response.

These context objects are powerful and are the key to advanced filter behaviors like short-circuiting the pipeline and handling exceptions. We'll make use of them in section 13.2 when we create more complex filter examples.

The async version of the resource filter requires implementing a single method, as shown in listing 13.2. As for the sync version, you're passed a `ResourceExecutingContext` object as an argument, and you're passed a delegate representing the remainder of the filter pipeline. You must call this delegate (asynchronously) to execute the remainder of the pipeline, which will return an instance of `ResourceExecutedContext`.

Listing 13.2 Example resource filter implementing `IAsyncResourceFilter`

```
public class LogAsyncResourceFilter : Attribute, IAsyncResourceFilter
{
    public async Task OnResourceExecutionAsync(
        ResourceExecutingContext context,
        ResourceExecutionDelegate next)
    {
        Console.WriteLine("Executing async!");
        ResourceExecutedContext executedContext = await next();
        Console.WriteLine("Executed async!");
    }
}
```

Executed at the start of the pipeline, after authorization filters

You're provided a delegate, which encapsulates the remainder of the filter pipeline.

Called before the rest of the pipeline executes

Executes the rest of the pipeline and obtains a ResourceExecutedContext instance

Called after the rest of the pipeline executes

The sync and async filter implementations have subtle differences, but for most purposes they're identical. I recommend implementing the sync version if possible, and only falling back to the async version if you need to.

You've created a couple of filters now, so we should look at how to use them in the application. In the next section we'll tackle two specific issues: how to control which requests execute your new filters, and how to control the order in which they execute.

13.1.5 Adding filters to your actions, controllers, Razor Pages, and globally

In section 13.1.2 I discussed the similarities and differences between middleware and filters. One of those differences is that filters can be scoped to specific actions or controllers, so that they only run for certain requests. Alternatively, you can apply a filter globally, so that it runs for every MVC action and Razor Page.

By adding filters in different ways, you can achieve several different results. Imagine you have a filter that forces you to log in to execute an action. How you add the filter to your app will significantly change your app's behavior:

- *Apply the filter to a single action or Razor Page*—Anonymous users could browse the app as normal, but if they tried to access the protected action or Razor Page, they would be forced to log in.
- *Apply the filter to a controller*—Anonymous users could access actions from other controllers, but accessing any action on the protected controller would force them to log in.
- *Apply the filter globally*—Users couldn't use the app without logging in. Any attempt to access an action or Razor Page would redirect the user to the login page.

NOTE ASP.NET Core comes with just such a filter out of the box: `AuthorizeFilter`. I'll discuss this filter in section 13.2.1, and you'll be seeing a lot more of it in chapter 15.

As I described in the previous section, you normally create filters as attributes, and for good reason—it makes it easy for you to apply them to MVC controllers, actions, and Razor Pages. In this section you'll see how to apply `LogResourceFilter` from listing 13.1 to an action, a controller, a Razor Page, and globally. The level at which the filter applies is called its *scope*.

DEFINITION The *scope* of a filter refers to how many different actions it applies to. A filter can be scoped to the action method, to the controller, to a Razor Page, or globally.

You'll start at the most specific scope—applying filters to a single action. The following listing shows an example of an MVC controller that has two action methods: one with `LogResourceFilter` and one without.

Listing 13.3 Applying filters to an action method

```
public class RecipeController : ControllerBase
{
    [LogResourceFilter]
    public IActionResult Index()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return OK();
    }
}
```

LogResourceFilter will run as part of the pipeline when executing this action.

This action method has no filters at the action level.

Alternatively, if you want to apply the same filter to every action method, you could add the attribute at the controller scope, as in the next listing. Every action method in the controller will use `LogResourceFilter`, without having to specifically decorate each method.

Listing 13.4 Applying filters to a controller

```
[LogResourceFilter]
public class RecipeController : ControllerBase
{
    public IActionResult Index ()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return Ok();
    }
}
```

The LogResourceFilter Is added to every action on the controller.

Every action in the controller is decorated with the filter.

For Razor Pages, you can apply attributes to your `PageModel`, as shown in the following listing. The filter applies to all page handlers in the Razor Page—it's not possible to apply filters to a single page handler; you must apply them at the page level.

Listing 13.5 Applying filters to a Razor Page

```
[LogResourceFilter]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }

    public void OnPost()
    {
    }
}
```

The LogResourceFilter Is added to the Razor Page's PageModel.

The filter applies to every page handler in the page.

Filters you apply as attributes to controllers, actions, and Razor Pages are automatically discovered by the framework when your application starts up. For common attributes, you can go one step further and apply filters globally, without having to decorate individual classes.

You add global filters in a different way than controller- or action-scoped filters—by adding a filter directly to the MVC services when configuring your controllers and Razor Pages in `Startup`. This listing shows three equivalent ways to add a globally scoped filter.

Listing 13.6 Applying filters globally to an application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers(options =>
        {
            options.Filters.Add(new LogResourceFilter());
            options.Filters.Add(typeof(LogResourceFilter));
            options.Filters.Add<LogResourceFilter>();
        });
    }
}
```

Adds filters using the MvcOptions object

You can pass an instance of the filter directly. . .

. . . or pass in the Type of the filter and let the framework create it.

Alternatively, the framework can create a global filter using a generic type parameter.

You can configure the `MvcOptions` by using the `AddControllers()` overload. When you configure filters globally, they apply both to controllers *and* to any Razor Pages in your application. If you're using Razor Pages in your application instead, there isn't an overload for configuring the `MvcOptions`. Instead you need to use the `AddMvcOptions()` extension method to configure the filters, as shown in the following listing.

Listing 13.7 Applying filters globally to a Razor Pages application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages()
            .AddMvcOptions(options =>
            {
                options.Filters.Add(new LogResourceFilter());
                options.Filters.Add(typeof(LogResourceFilter));
                options.Filters.Add<LogResourceFilter>();
            });
    }
}
```

This method doesn't let you pass a lambda to configure the MvcOptions.

You must use an extension method to add the filters to the MvcOptions object.

You can configure the filters in any of the ways shown previously.

With potentially three different scopes in play, you'll often find action methods that have multiple filters applied to them: some applied directly to the action method, and others inherited from the controller or globally. The question then becomes, which filter runs first?

13.1.6 Understanding the order of filter execution

You've seen that the filter pipeline contains five different *stages*, one for each type of filter. These stages always run in the fixed order I described in sections 13.1.1 and 13.1.2. But within each stage, you can also have multiple filters of the same type (for example, multiple resource filters) that are part of a single action method's pipeline. These could all have multiple *scopes*, depending on how you added them, as you saw in the last section.

In this section, we're thinking about the *order of filters within a given stage* and how scope affects this. We'll start by looking at the default order and then move on to ways to customize the order to your own requirements.

The default scope execution order

When thinking about filter ordering, it's important to remember that resource, action, and result filters implement two methods: an `*Executing` before method and an `*Executed` after method. On top of that, page filters implement three methods! The order in which each method executes depends on the scope of the filter, as shown in figure 13.4 for the resource filter stage.

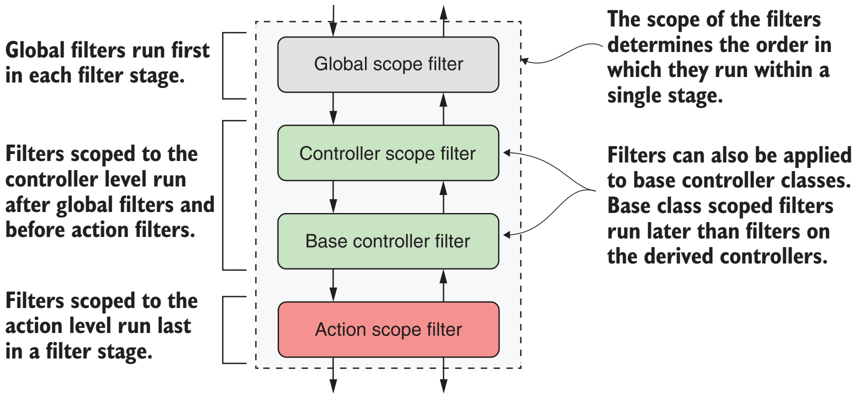

Global filters run first in each filter stage.
Filters scoped to the controller level run after global filters and before action filters.
Filters scoped to the action level run last in a filter stage.
Global scope filter
Controller scope filter
Base controller filter
Action scope filter
The scope of the filters determines the order in which they run within a single stage.
Filters can also be applied to base controller classes. Base class scoped filters run later than filters on the derived controllers.

Figure 13.4 The default filter ordering within a given stage, based on the scope of the filters. For the `*Executing` method, globally scoped filters run first, followed by controller-scoped, and finally, action-scoped filters. For the `*Executed` method, the filters run in reverse order.

By default, filters execute from the broadest scope (global) to the narrowest (action) when running the `*Executing` method for each stage. The filters' `*Executed` methods run in reverse order, from the narrowest scope (action) to the broadest (global).

The ordering for Razor Pages is somewhat simpler, given that you only have two scopes—global scope filters and Razor Page scope filters. For Razor Pages, global scope filters run the `*Executing` and `PageHandlerSelected` methods first, followed by the page scope filters. For the `*Executed` methods, the filters run in reverse order.

You'll sometimes find you need a bit more control over this order, especially if you have, for example, multiple action filters applied at the same scope. The filter pipeline caters to this requirement by way of the `IOrderedFilter` interface.

OVERRIDING THE DEFAULT ORDER OF FILTER EXECUTION WITH IORDEREDFILTER

Filters are great for extracting cross-cutting concerns from your controller actions and Razor Page, but if you have multiple filters applied to an action, you'll often need to control the precise order in which they execute.

Scope can get you some of the way, but for those other cases, you can implement `IOrderedFilter`. This interface consists of a single property, `Order`:

```
public interface IOrderedFilter
{
    int Order { get; }
}
```

You can implement this property in your filters to set the order in which they execute. The filter pipeline orders the filters in a given stage based on this value first, from lowest to highest, and uses the default scope order to handle ties, as shown in figure 13.5.

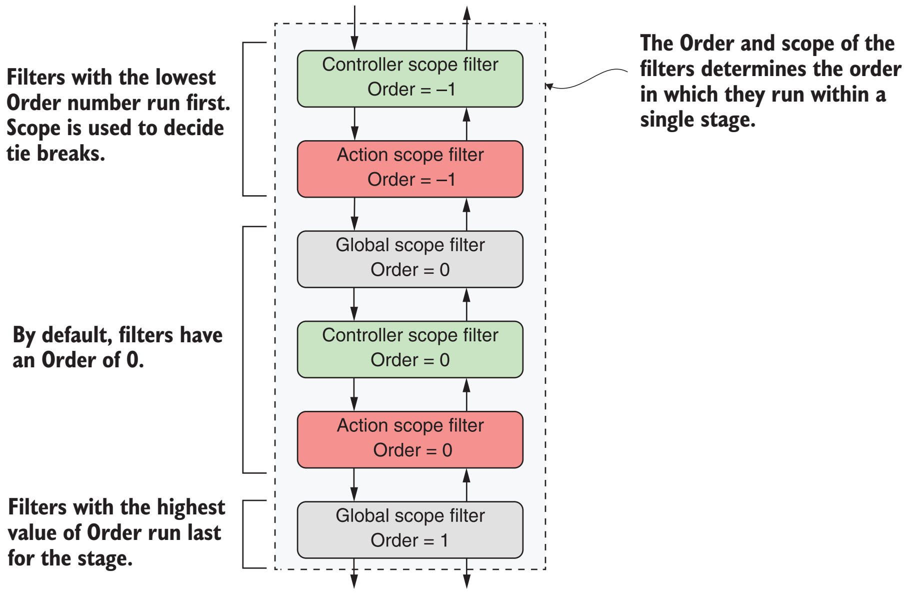

Filters with the lowest Order number run first. Scope is used to decide tie breaks.
By default, filters have an Order of 0.
Filters with the highest value of Order run last for the stage.
The Order and scope of the filters determines the order in which they run within a single stage.
Controller scope filter Order = –1
Action scope filter Order = –1
Global scope filter Order = 0
Controller scope filter Order = 0
Action scope filter Order = 0
Global scope filter Order = 1

Figure 13.5 Controlling the filter order for a stage using the `IOrderedFilter` interface. Filters are ordered by the `Order` property first, and then by scope.

The filters for `Order = -1` execute first, as they have the lowest `Order` value. The controller filter executes first because it has a broader scope than the action-scope filter. The filters with `Order = 0` execute next, in the default scope order, as shown in figure 13.5. Finally, the filter with `Order = 1` executes.

By default, if a filter doesn't implement `IOrderedFilter`, it's assumed to have `Order = 0`. All the filters that ship as part of ASP.NET Core have `Order = 0`, so you can implement your own filters relative to these.

This section has covered most of the technical details you need to use filters and create custom implementations for your own application. In the next section you'll see some of the built-in filters provided by ASP.NET Core, as well as some practical examples of filters you might want to use in your own applications.

13.2 Creating custom filters for your application

ASP.NET Core includes a number of filters that you can use, but often the most useful filters are the custom ones that are specific to your own apps. In this section we'll work through each of the six types of filters. I'll explain in more detail what they're for and when you should use them. I'll point out examples of these filters that are part of ASP.NET Core itself, and you'll see how to create custom filters for an example application.

To give you something realistic to work with, we'll start with a Web API controller for accessing the recipe application from chapter 12. This controller contains two actions: one for fetching a `RecipeDetailViewModel` and another for updating a `Recipe` with new values. This listing shows your starting point for this chapter, including both of the action methods.

Listing 13.8 Recipe Web API controller before refactoring to use filters

```
[Route("api/recipe")]
public class RecipeApiController : ControllerBase
{
    private const bool IsEnabled = true;
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {
        if (!IsEnabled) { return BadRequest(); }
        try
        {
            if (!_service.DoesRecipeExist(id))
            {
                return NotFound();
            }
            var detail = _service.GetRecipeDetail(id);
```

This field would be passed in as configuration and is used to control access to actions.

If the API isn't enabled, block further execution.

If the requested Recipe doesn't exist, return a 404 response.

Fetch RecipeDetail ViewModel.

```
            Response.GetTypedHeaders().LastModified =
                detail.LastModified;
            return Ok(detail);
        }
        catch (Exception ex)
        {
            return GetErrorResponse(ex);
        }
    }

    [HttpPost("{id}")]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        if (!IsEnabled) { return BadRequest(); }
        try
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            if (!_service.DoesRecipeExist(id))
            {
                return NotFound();
            }
            _service.UpdateRecipe(command);
            return Ok();
        }
        catch (Exception ex)
        {
            return GetErrorResponse(ex);
        }
    }

    private static IActionResult GetErrorResponse(Exception ex)
    {
        var error = new ProblemDetails
        {
            Title = "An error occurred",
            Detail = context.Exception.Message,
            Status = 500,
            Type = "https://httpstatuses.com/500"
        };

        return new ObjectResult(error)
        {
            StatusCode = 500
        };
    }
}
```

Sets the Last-Modified response header to the value in the model

Returns the view model with a 200 response

If an exception occurs, catch it and return the error in an expected format, as a 500 error.

If the API isn't enabled, block further execution.

Validate the binding model and return a 400 response if there are errors.

If the requested Recipe doesn't exist, return a 404 response.

Update the Recipe from the command and return a 200 response.

If an exception occurs, catch it and return the error in an expected format, as a 500 error.

These action methods currently have a *lot* of code to them, which hides the intent of each action. There's also quite a lot of duplication between the methods, such as checking that the `Recipe` entity exists and formatting exceptions.

In this section you're going to refactor this controller to use filters for all the code in the methods that's unrelated to the intent of each action. By the end of the chapter, you'll have a much simpler controller that's far easier to understand, as shown here.

Listing 13.9 Recipe Web API controller after refactoring to use filters

```
[Route("api/recipe")]
[ValidateModel, HandleException, FeatureEnabled(IsEnabled = true)]
public class RecipeApiController : ControllerBase
{
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}"), EnsureRecipeExists, AddLastModifiedHeader]
    public IActionResult Get(int id)
    {
        var detail = _service.GetRecipeDetail(id);
        return Ok(detail);
    }

    [HttpPost("{id}"), EnsureRecipeExists]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        _service.UpdateRecipe(command);
        return Ok();
    }
}
```

The filters encapsulate the majority of logic common to multiple action methods.

Placing filters at the action level limits them to a single action.

The intent of the action, return a Recipe view model, is much clearer.

Placing filters at the action level can control the order in which they execute.

The intent of the action, update a Recipe, is much clearer.

I think you'll have to agree that the controller in listing 13.9 is much easier to read! In this section, you'll refactor the controller bit by bit, removing cross-cutting code to get to something more manageable. All the filters we'll create in this section will use the sync filter interfaces—I'll leave it to you, as an exercise, to create their async counterparts. We'll start by looking at authorization filters and how they relate to security in ASP.NET Core.

13.2.1 Authorization filters: Protecting your APIs

Authentication and *authorization* are related, fundamental concepts in security that we'll be looking at in detail in chapters 14 and 15.

DEFINITION *Authentication* is concerned with determining *who* made a request. *Authorization* is concerned with *what* a user is allowed to access.

Authorization filters run first in the MVC filter pipeline, before any other filters. They control access to the action method by immediately short-circuiting the pipeline when a request doesn't meet the necessary requirements.

ASP.NET Core has a built-in authorization framework that you should use when you need to protect your MVC application or your Web APIs. You can configure this framework with custom policies that let you finely control access to your actions.

TIP It's possible to write your own authorization filters by implementing `IAuthorizationFilter` or `IAsyncAuthorizationFilter`, but I strongly advise against it. The ASP.NET Core authorization framework is highly configurable and should meet all your needs.

At the heart of the ASP.NET Core authorization framework is an authorization filter, `AuthorizeFilter`, which you can add to the filter pipeline by decorating your actions or controllers with the `[Authorize]` attribute. In its simplest form, adding the `[Authorize]` attribute to an action, as in the following listing, means the request must be made by an authenticated user to be allowed to continue. If you're not logged in, it will short-circuit the pipeline, returning a `401 Unauthorized` response to the browser.

Listing 13.10 Adding `[Authorize]` to an action method

```
public class RecipeApiController : ControllerBase
{
    public IActionResult Get(int id)
    {
        // method body
    }

    [Authorize]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        // method body
    }
}
```

The Get method has no [Authorize] attribute, so it can be executed by anyone.

Adds the AuthorizeFilter to the filter pipeline using [Authorize]

The Edit method can only be executed if you're logged in.

As with all filters, you can apply the `[Authorize]` attribute at the controller level to protect all the actions on a controller, to a Razor Page to protect all the page handler methods in a page, or even globally to protect every endpoint in your app.

NOTE We'll explore authorization in detail in chapter 15, including how to add more detailed requirements, so that only specific sets of users can execute an action.

The next filters in the pipeline are resource filters. In the next section you'll extract some of the common code from `RecipeApiController` and see how easy it is to create a short-circuiting filter.

13.2.2 Resource filters: Short-circuiting your action methods

Resource filters are the first general-purpose filters in the MVC filter pipeline. In section 13.1.4 you saw minimal examples of both sync and async resource filters, which logged to the console. In your own apps, you can use resource filters for a wide range of purposes, thanks to the fact that they execute so early (and late) in the filter pipeline.

The ASP.NET Core framework includes a few different implementations of resource filters you can use in your apps:

- `ConsumesAttribute`—Can be used to restrict the allowed formats an action method can accept. If your action is decorated with `[Consumes("application/json")]` but the client sends the request as XML, the resource filter will short-circuit the pipeline and return a `415 Unsupported Media Type` response.
- `DisableFormValueModelBindingAttribute`—This filter prevents model binding from binding to form data in the request body. This can be useful if you know an action method will be handling large file uploads that you need to manage manually. The resource filters run before model binding, so you can disable the model binding for a single action in this way.[1]

Resource filters are useful when you want to ensure the filter runs early in the pipeline, before model binding. They provide an early hook into the pipeline for your logic, so you can quickly short-circuit the request if you need to.

Look back at listing 13.8 and see if you can refactor any of the code into a resource filter. One candidate line appears at the start of both the `Get` and `Edit` methods:

```
if (!IsEnabled) { return BadRequest(); }
```

This line of code is a *feature toggle* that you can use to disable the availability of the whole API, based on the `IsEnabled` field. In practice, you'd probably load the `IsEnabled` field from a database or configuration file so you could control the availability dynamically at runtime, but for this example I'm using a hardcoded value.[2]

This piece of code is self-contained cross-cutting logic, which is somewhat tangential to the main intent of each action method—a perfect candidate for a filter. You want to execute the feature toggle early in the pipeline, before any other logic, so a resource filter makes sense.

TIP Technically, you could also use an authorization filter for this example, but I'm following my own advice of "Don't write your own Authorization filters!"

The next listing shows an implementation of `FeatureEnabledAttribute`, which extracts the logic from the action methods and moves it into the filter. I've also exposed the `IsEnabled` field as a property on the filter.

[1] For details on handling file uploads, see Microsoft's documentation: http://mng.bz/4Z2D.

[2] To read more about using feature toggles in your applications, see my series: "Adding feature flags to an ASP.NET Core app" blog entry: http://mng.bz/2e40.

Listing 13.11 The `FeatureEnabledAttribute` resource filter

```
public class FeatureEnabledAttribute : Attribute, IResourceFilter
{
    public bool IsEnabled { get; set; }
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        if (!IsEnabled)
        {
            context.Result = new BadRequestResult();
        }
    }
    public void OnResourceExecuted(
        ResourceExecutedContext context) { }
}
```

Defines whether the feature is enabled

Executes before model binding, early in the filter pipeline

If the feature isn't enabled, short-circuits the pipeline by setting the context.Result property

Must be implemented to satisfy IResourceFilter, but not needed in this case

This simple resource filter demonstrates a number of important concepts, which are applicable to most filter types:

- The filter is an attribute as well as a filter. This lets you decorate your controller, action methods, and Razor Pages with it using `[FeatureEnabled(IsEnabled = true)]`.
- The filter interface consists of two methods: `*Executing`, which runs before model binding, and `*Executed`, which runs after the result has been executed. You must implement both, even if you only need one for your use case.
- The filter execution methods provide a `context` object. This provides access to, among other things, the `HttpContext` for the request and metadata about the action method the middleware will execute.
- To short-circuit the pipeline, set the `context.Result` property to an `IActionResult` instance. The framework will execute this result to generate the response, bypassing any remaining filters in the pipeline and skipping the action method (or page handler) entirely. In this example, if the feature isn't enabled, you bypass the pipeline by returning `BadRequestResult`, which will return a 400 error to the client.

By moving this logic into the resource filter, you can remove it from your action methods and instead decorate the whole API controller with a simple attribute:

```
[Route("api/recipe"), FeatureEnabled(IsEnabled = true)]
public class RecipeApiController : ControllerBase
```

You've only extracted two lines of code from your action methods so far, but you're on the right track. In the next section we'll move on to action filters and extract two more filters from the action method code.

13.2.3 Action filters: Customizing model binding and action results

Action filters run just after model binding, before the action method executes. Thanks to this positioning, action filters can access all the arguments that will be used to execute the action method, which makes them a powerful way of extracting common logic out of your actions.

On top of this, they also run just after the action method has executed and can completely change or replace the `IActionResult` returned by the action if you want. They can even handle exceptions thrown in the action.

NOTE Action filters don't execute for Razor Pages. Similarly, page filters don't execute for action methods.

The ASP.NET Core framework includes several action filters out of the box. One of these commonly used filters is `ResponseCacheFilter`, which sets HTTP caching headers on your action-method responses.

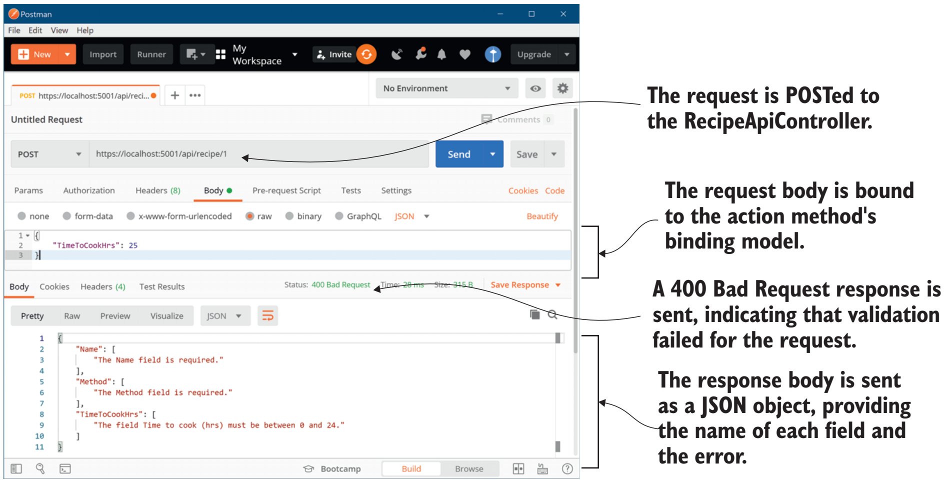
TIP Caching is a broad topic that aims to improve the performance of an application over the naive approach. But caching can also make debugging issues difficult and may even be undesirable in some situations. Consequently, I often apply `ResponseCacheFilter` to my action methods to set HTTP caching headers that *disable* caching! You can read about this and other approaches to caching in Microsoft's "Response caching in ASP.NET Core" documentation at http://mng.bz/2eGd.

The real power of action filters comes when you build filters tailored to your own apps by extracting common code from your action methods. To demonstrate, I'm going to create two custom filters for `RecipeApiController`:

- `ValidateModelAttribute`—This will return `BadRequestResult` if the model state indicates that the binding model is invalid and will short-circuit the action execution. This attribute used to be a staple of my Web API applications, but the `[ApiController]` attribute now handles this (and more) for you. Nevertheless, I think it's useful to understand what's going on behind the scenes.
- `EnsureRecipeExistsAttribute`—This will use each action method's `id` argument to validate that the requested `Recipe` entity exists before the action method runs. If the `Recipe` doesn't exist, the filter will return `NotFoundResult` and will short-circuit the pipeline.

As you saw in chapter 6, the MVC framework automatically validates your binding models before executing your actions, but it's up to you to decide what to do about it. For Web API controllers, it's common to return a `400 Bad Request` response containing a list of the errors, as shown in figure 13.6.

You should ordinarily use the `[ApiController]` attribute on your Web API controllers, which gives you this behavior automatically. But if you can't, or don't want to use that attribute, you can create a custom action filter instead. Listing 13.12



```
{
    "TimeToCookHrs": 25
}
```

```
{
    "Name": [
        "The Name field is required."
    ],
    "Method": [
        "The Method field is required."
    ],
    "TimeToCookHrs": [
        "The field Time to cook (hrs) must be between 0 and 24."
    ]
}
```

Figure 13.6 Posting data to a Web API using Postman. The data is bound to the action method's binding model and validated. If validation fails, it's common to return a `400 Bad Request` response with a list of the validation errors.

shows a basic implementation that is similar to the behavior you get with the `[Api-Controller]` attribute.

Listing 13.12 The action filter for validating `ModelState`

For convenience, you derive from the ActionFilterAttribute base class.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result =
                new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

Overrides the Executing method to run the filter before the Action executes

Model binding and validation have already run at this point, so you can check the state.

If the model isn't valid, set the Result property; this short-circuits the action execution.

This attribute is self-explanatory and follows a similar pattern to the resource filter in section 13.2.2, but with a few interesting points:

- I have derived from the abstract `ActionFilterAttribute`. This class implements `IActionFilter` and `IResultFilter`, as well as their async counterparts, so you can override the methods you need as appropriate. This avoids needing to add an unused `OnActionExecuted()` method, but using the base class is entirely optional and a matter of preference.

- Action filters run after model binding has taken place, so `context.ModelState` contains the validation errors if validation failed.
- Setting the `Result` property on `context` short-circuits the pipeline. But due to the position of the action filter stage, only the action method execution and later action filters are bypassed; all the other stages of the pipeline run as though the action had executed as normal.

If you apply this action filter to your `RecipeApiController`, you can remove this code from the start of both the action methods, as it will run automatically in the filter pipeline:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

You'll use a similar approach to remove the duplicate code that checks whether the `id` provided as an argument to the action methods corresponds to an existing Recipe entity.

The following listing shows the `EnsureRecipeExistsAttribute` action filter. This uses an instance of `RecipeService` to check whether the `Recipe` exists and returns a `404 Not Found` if it doesn't.

Listing 13.13 An action filter to check whether a `Recipe` exists

```
public class EnsureRecipeExistsAtribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext
            .RequestServices.GetService(typeof(RecipeService));
        var recipeId = (int) context.ActionArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }
}
```

Fetches an instance of RecipeService from the DI container

Retrieves the id parameter that will be passed to the action method when it executes

Checks whether a Recipe entity with the given RecipeId exists

If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline

As before, you've derived from `ActionFilterAttribute` for simplicity and overridden the `OnActionExecuting` method. The main functionality of the filter relies on the `DoesRecipeExist()` method of `RecipeService`, so the first step is to obtain an instance of `RecipeService`. The `context` parameter provides access to the `HttpContext` for the request, which in turn lets you access the DI container and use `RequestServices.GetService()` to return an instance of `RecipeService`.

WARNING This technique for obtaining dependencies is known as *service location* and is generally considered an antipattern.[3] In section 13.4 I'll show you a better way to use the DI container to inject dependencies into your filters.

As well as `RecipeService`, the other piece of information you need is the `id` argument of the `Get` and `Edit` action methods. In action filters, model binding has already occurred, so the arguments that the framework will use to execute the action method are already known and are exposed on `context.ActionArguments`.

The action arguments are exposed as `Dictionary<string, object>`, so you can obtain the `id` parameter using the `"id"` `string` key. Remember to cast the object to the correct type.

TIP Whenever I see magic strings like this, I always try to replace them by using the `nameof` operator. Unfortunately, `nameof` won't work for method arguments like this, so be careful when refactoring your code. I suggest explicitly applying the action filter to the action method (instead of globally, or to a controller) to remind you about that implicit coupling.

With `RecipeService` and `id` in place, it's a case of checking whether the identifier corresponds to an existing `Recipe` entity and, if not, setting `context.Result` to `NotFoundResult`. This short-circuits the pipeline and bypasses the action method altogether.

NOTE Remember, you can have multiple action filters running in a single stage. Short-circuiting the pipeline by setting `context.Result` will prevent later filters in the stage from running, as well as bypass the action method execution.

Before we move on, it's worth mentioning a special case for action filters. The `ControllerBase` base class implements `IActionFilter` and `IAsyncActionFilter` itself. If you find yourself creating an action filter for a single controller and you want to apply it to every action in that controller, you can override the appropriate methods on your controller.

Listing 13.14 Overriding action filter methods directly on `ControllerBase`

```
public class HomeController : ControllerBase        <- Derives from the ControllerBase class
{
    public override void OnActionExecuting(          | Runs before any other action filters
        ActionExecutingContext context)              | for every action in the controller
    { }
    public override void OnActionExecuted(           | Runs after all other action filters
        ActionExecutedContext context)               | for every action in the controller
    { }
}
```

[3] For a detailed discussion of DI patterns and antipatterns, see chapter 5 of *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019) http://mng.bz/RXOK.

If you override these methods on your controller, they'll run in the action filter stage of the filter pipeline for every action on the controller. The `OnActionExecuting` `ControllerBase` method runs before any other action filters, regardless of ordering or scope, and the `OnActionExecuted` method runs after all other action filters.

> **TIP** The controller implementation can be useful in some cases, but you can't control the ordering related to other filters. Personally, I generally prefer to break logic into explicit, declarative filter attributes but, as always, the choice is yours.

With the resource and action filters complete, your controller is looking much tidier, but there's one aspect in particular that would be nice to remove: the exception handling. In the next section, we'll look at how to create a custom exception filter for your controller, and why you might want to do this instead of using exception handling middleware.

13.2.4 Exception filters: Custom exception handling for your action methods

In chapter 3 I went into some depth about types of error-handling middleware you can add to your apps. These let you catch exceptions thrown from any later middleware and handle them appropriately. If you're using exception handling middleware, you may be wondering why we need exception filters at all.

The answer to this is pretty much the same as I outlined in section 13.1.3: filters are great for cross-cutting concerns, when you need behavior that's either specific to MVC or that should only apply to certain routes.

Both of these can apply in exception handling. Exception filters are part of the MVC framework, so they have access to the context in which the error occurred, such as the action or Razor Page that was executing. This can be useful for logging additional details when errors occur, such as the action parameters that caused the error.

> **WARNING** If you use exception filters to record action method arguments, make sure you're not storing sensitive data in your logs, such as passwords or credit card details.

You can also use exception filters to handle errors from different routes in different ways. Imagine you have both Razor Pages and Web API controllers in your app, as we do in the recipe app. What happens when an exception is thrown by a Razor Page?

As you saw in chapter 3, the exception travels back up the middleware pipeline and is caught by exception handler middleware. The exception handler middleware will re-execute the pipeline and generate an HTML error page.

That's great for your Razor Pages, but what about exceptions in your Web API controllers? If your API throws an exception, and consequently returns HTML generated by the exception handler middleware, that's going to break a client that has called the API expecting a JSON response!

Instead, exception filters let you handle the exception in the filter pipeline and generate an appropriate response body. The exception handler middleware only intercepts errors without a body, so it will let the modified Web API response pass untouched.

NOTE The `[ApiController]` attribute converts error `StatusCodeResults` to a `ProblemDetails` object, but it *doesn't* catch exceptions.

Exception filters can catch exceptions from more than just your action methods and page handlers. They'll run if an exception occurs at these times:

- During model binding or validation
- When the action method or page handler is executing
- When an action filter or page filter is executing

You should note that exception filters won't catch exceptions thrown in any filters other than action and page filters, so it's important that your resource and result filters don't throw exceptions. Similarly, they won't catch exceptions thrown when executing an `IActionResult`, such as when rendering a Razor view to HTML.

Now that you know why you might want an exception filter, go ahead and implement one for `RecipeApiController`, as shown next. This lets you safely remove the `try-catch` block from your action methods, knowing that your filter will catch any errors.

Listing 13.15 The `HandleExceptionAttribute` exception filter

```
public class HandleExceptionAttribute : ExceptionFilterAttribute
{
    public override void OnException(ExceptionContext context)
    {
        var error = new ProblemDetails
        {
            Title = "An error occurred",
            Detail = context.Exception.Message,
            Status = 500,
            Type = "https://httpstatuses.com/500"
        };

        context.Result = new ObjectResult(error)
        {
            StatusCode = 500
        };
        context.ExceptionHandled = true;
    }
}
```

ExceptionFilterAttribute is an abstract base class that implements IExceptionFilter.

There's only a single method to override for IExceptionFilter.

Building a problem details object to return in the response

Creates an ObjectResult to serialize the ProblemDetails and to set the response status code

Marks the exception as handled to prevent it propagating into the middleware pipeline

It's quite common to have an exception filter in your application, especially if you are mixing API controllers and Razor Pages in your application, but they're not always necessary. If you can handle all the exceptions in your application with a single piece of middleware, then ditch the exception filters and go with that instead.

You're almost done refactoring your `RecipeApiController`. You just have one more filter type to add: result filters. Custom result filters tend to be relatively rare in the apps I've written, but they have their uses, as you'll see.

13.2.5 Result filters: Customizing action results before they execute

If everything runs successfully in the pipeline, and there's no short-circuiting, the next stage of the pipeline, after action filters, is result filters. These run just before and after the `IActionResult` returned by the action method (or action filters) is executed.

> **WARNING** If the pipeline is short-circuited by setting `context.Result`, the result filter stage won't be run, but `IActionResult` will still be executed to generate the response. The exceptions to this rule are action and page filters—these only short-circuit the action execution, as you saw in figures 13.2 and 13.3, so result filters run as normal, as though the action or page handler itself generated the response.

Result filters run immediately after action filters, so many of their use cases are similar, but you typically use result filters to customize the way the `IActionResult` executes. For example, ASP.NET Core has several result filters built into its framework:

- `ProducesAttribute`—This forces a Web API result to be serialized to a specific output format. For example, decorating your action method with `[Produces("application/xml")]` forces the formatters to try to format the response as XML, even if the client doesn't list XML in its `Accept` header.
- `FormatFilterAttribute`—Decorating an action method with this filter tells the formatter to look for a route value or query string parameter called `format`, and to use that to determine the output format. For example, you could call `/api/recipe/11?format=json` and `FormatFilter` will format the response as JSON, or call `api/recipe/11?format=xml` and get the response as XML.[4]

As well as controlling the output formatters, you can use result filters to make any last-minute adjustments before `IActionResult` is executed and the response is generated.

As an example of the kind of flexibility available, in the following listing I demonstrate setting the `LastModified` header, based on the object returned from the action. This is a somewhat contrived example—it's specific enough to a single action that it doesn't warrant being moved to a result filter—but hopefully you get the idea.

[4] Remember, you need to explicitly configure the XML formatters if you want to serialize to XML. For details on formatting results based on the URL, see my blog entry on the topic: http://mng.bz/1rYV.

Listing 13.16 Setting a response header in a result filter

Checks whether the action result returned a 200 Ok result with a view model.

ResultFilterAttribute provides a useful base class you can override.

```
public class AddLastModifedHeaderAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(
        ResultExecutingContext context)
    {
        if (context.Result is OkObjectResult result
            && result.Value is RecipeDetailViewModel detail)
        {
            var viewModelDate = detail.LastModified;
            context.HttpContext.Response
              .GetTypedHeaders().LastModified = viewModelDate;
        }
    }
}
```

You could also override the Executed method, but the response would already be sent by then.

Checks whether the view model type is Recipe-DetailView-Model . . .

. . . if it is, fetches the LastModified property and sets the Last-Modified header in the response

I've used another helper base class here, `ResultFilterAttribute`, so you only need to override a single method to implement the filter. Fetch the current `IActionResult`, exposed on `context.Result`, and check that it's an `OkObjectResult` instance with a `RecipeDetailViewModel` value. If it is, fetch the `LastModified` field from the view model and add a `Last-Modified` header to the response.

TIP `GetTypedHeaders()` is an extension method that provides strongly typed access to request and response headers. It takes care of parsing and formatting the values for you. You can find it in the Microsoft.AspNetCore.Http namespace.

As with resource and action filters, result filters can implement a method that runs *after* the result has been executed: `OnResultExecuted`. You can use this method, for example, to inspect exceptions that happened during the execution of `IActionResult`.

WARNING Generally, you can't modify the response in the `OnResultExecuted` method, as you may have already started streaming the response to the client.

Running result filters after short-circuits with `IAlwaysRunResultFilter`

Result filters are designed to "wrap" the execution of an `IActionResult` returned by an action method or action filter so that you can customize how the action result is executed. However, this customization doesn't apply to `IActionResults` set when you short-circuit the filter pipeline by setting `context.Result` in an authorization filter, resource filter, or exception filter.

That's often not a problem, as many result filters are designed to handle "happy path" transformations. But sometimes you want to make sure a transformation is *always* applied to an `IActionResult`, regardless of whether it was returned by an action method or a short-circuiting filter.

For those cases, you can implement `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter`. These interfaces extend (and are identical) to the standard result filter interfaces, so they run just like normal result filters in the filter pipeline. But these interfaces mark the filter to *also* run after an authorization filter, resource filter, or exception filter short-circuits the pipeline, where standard result filters *won't* run.

You can use `IAlwaysRunResultFilter` to ensure that certain action results are always updated. For example, the documentation shows how to use an `IAlwaysRunResultFilter` to convert a `415 StatusCodeResult` into a `422 StatusCodeResult`, regardless of the source of the action result. See the "IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter" section of Microsoft's "Filters in ASP.NET Core" documentation: http://mng.bz/JDo0.

We've finished simplifying the `RecipeApiController` now. By extracting various pieces of functionality to filters, the original controller in listing 13.8 has been simplified to the version in listing 13.9. This is obviously a somewhat extreme and contrived demonstration, and I'm not advocating that filters should always be your go-to option.

TIP Filters should be a last resort in most cases. Where possible, it is often preferable to use a simple private method in a controller, or to push functionality into the domain instead of using filters. Filters should generally be used to extract repetitive, HTTP-related, or common cross-cutting code from your controllers.

There's still one more filter we haven't looked at yet, because it only applies to Razor Pages: page filters.

13.2.6 Page filters: Customizing model binding for Razor Pages

As already discussed, action filters only apply to controllers and actions; they have no effect on Razor Pages. Similarly, page filters have no effect on controllers and actions. Nevertheless, page filters and action filters fulfill similar roles.

As is the case for action filters, the ASP.NET Core framework includes several page filters out of the box. One of these is the Razor Page equivalent of the caching action filter, `ResponseCacheFilter`, called `PageResponseCacheFilter`. This works identically to the action-filter equivalent I described in section 13.2.3, setting HTTP caching headers on your Razor Page responses.

Page filters are somewhat unusual, as they implement three methods, as discussed in section 13.1.2. In practice, I've rarely seen a page filter that implements all three. It's unusual to need to run code immediately after page handler selection and before model validation. It's far more common to perform a role directly analogous to action filters.

For example, the following listing shows a page filter equivalent to the `EnsureRecipeExistsAttribute` action filter.

Listing 13.17 A page filter to check whether a `Recipe` exists

Fetches an instance of RecipeService from the DI container

Implement IPageFilter and as an attribute so you can decorate the Razor Page PageModel.

```
public class PageEnsureRecipeExistsAtribute : Attribute, IPageFilter
{
    public void OnPageHandlerSelected(
        PageHandlerSelectedContext context)
    {}

    public void OnPageHandlerExecuting(
        PageHandlerExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext
            .RequestServices.GetService(typeof(RecipeService));
        var recipeId = (int) context.HandlerArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }

    public void OnPageHandlerExecuted(
        PageHandlerExecutedContext context)
    { }
}
```

Executed after handler selection, before model binding—not used in this example

Executed after model binding and validation, before page handler execution

Checks whether a Recipe entity with the given RecipeId exists

If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline

Executed after page handler execution (or short-circuiting)—not used in this example

Retrieves the id parameter that will be passed to the page handler method when it executes

The page filter is very similar to the action filter equivalent. The most obvious difference is the need to implement three methods to satisfy the `IPageFilter` interface. You'll commonly want to implement the `OnPageHandlerExecuting` method, which runs just after model binding and validation, and before the page handler executes.

A subtle difference between the action filter code and the page filter code is that the action filter accesses the model-bound action arguments using `context.ActionArguments`. The page filter uses `context.HandlerArguments` in the example, but there's also another option.

Remember from chapter 6 that Razor Pages often bind to public properties on the `PageModel` using the `[BindProperty]` attribute. You can access those properties directly, instead of having to use magic strings, by casting a `HandlerInstance` property to the correct `PageModel` type, and accessing the property directly. For example,

```
var recipeId = ((ViewRecipePageModel)context.HandlerInstance).Id
```

Just as the `ControllerBase` class implements `IActionFilter`, so `PageModel` implements `IPageFilter` and `IAsyncPageFilter`. If you want to create an action filter for a single Razor Page, you could save yourself the trouble of creating a separate page filter and override these methods directly in your Razor Page.

TIP I generally find it's not worth the hassle of using page filters unless you have a *very* common requirement. The extra level of indirection page filters add, coupled with the typically bespoke nature of individual Razor Pages, means I normally find they aren't worth using. Your mileage may vary of course, but don't jump to them as a first option.

That brings us to the end of this detailed look at each of the filters in the MVC pipeline. Looking back and comparing listings 13.8 and 13.9, you can see filters allowed us to refactor the controllers and make the intent of each action method much clearer. Writing your code in this way makes it easier to reason about, as each filter and action has a single responsibility.

In the next section, we'll take a slight detour into exactly what happens when you short-circuit a filter. I've described *how* to do this, by setting the `context.Result` property on a filter, but I haven't yet described exactly what happens. For example, what if there are multiple filters in the stage when it's short-circuited? Do those still run?

13.3 Understanding pipeline short-circuiting

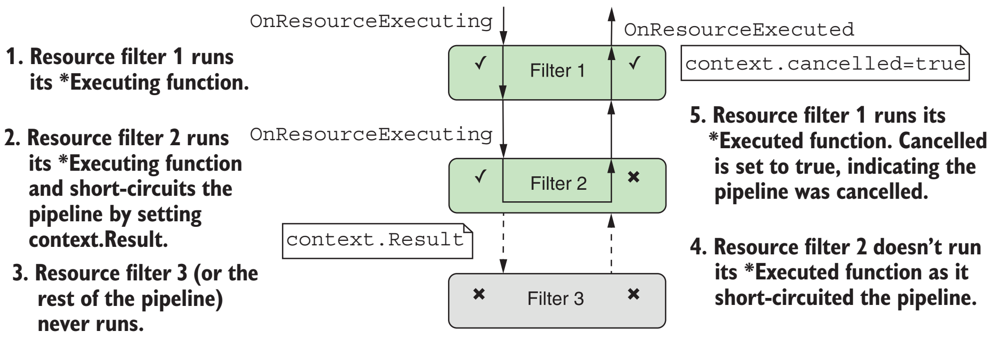
In this short section you'll learn about the details of filter-pipeline short-circuiting. You'll see what happens to the other filters in a stage when the pipeline is short-circuited, and how to short-circuit each type of filter.

A brief warning: the topic of filter short-circuiting can be a little confusing. Unlike middleware short-circuiting, which is cut-and-dried, the filter pipeline is a bit more nuanced. Luckily, you won't often find you need to dig into it, but when you do, you'll be glad of the detail.

You short-circuit the authorization, resource, action, page, and result filters by setting `context.Result` to `IActionResult`. Setting an action result in this way causes some, or all, of the remaining pipeline to be bypassed. But the filter pipeline isn't entirely linear, as you saw in figures 13.2 and 13.3, so short-circuiting doesn't always do an about-face back down the pipeline. For example, short-circuited action filters only bypass action method execution—the result filters and result execution stages still run.

The other difficultly is what happens if you have more than one type of filter. Let's say you have three resource filters executing in a pipeline. What happens if the second filter causes a short circuit? Any remaining filters are bypassed, but the first resource filter has already run its `*Executing` command, as shown in figure 13.7. This earlier filter gets to run its `*Executed` command too, with `context.Cancelled = true`, indicating that a filter in that stage (the resource filter stage) short-circuited the pipeline.

Understanding which other filters will run when you short-circuit a filter can be somewhat of a chore, but I've summarized each filter in table 13.1. You'll also find it useful to refer to figures 13.2 and 13.3 to visualize the shape of the pipeline when thinking about short-circuits.



OnResourceExecuting
OnResourceExecuted
1. Resource filter 1 runs its *Executing function.
Filter 1
context.cancelled=true
2. Resource filter 2 runs its *Executing function and short-circuits the pipeline by setting context.Result.
OnResourceExecuting
Filter 2
5. Resource filter 1 runs its *Executed function. Cancelled is set to true, indicating the pipeline was cancelled.
context.Result
3. Resource filter 3 (or the rest of the pipeline) never runs.
Filter 3
4. Resource filter 2 doesn't run its *Executed function as it short-circuited the pipeline.

Figure 13.7 The effect of short-circuiting a resource filter on other resource filters in that stage. Later filters in the stage won't run at all, but earlier filters run their `OnResourceExecuted` function.

Table 13.1 The effect of short-circuiting filters on filter-pipeline execution

Filter type	How to short-circuit?	What else runs?
Authorization filters	Set `context.Result`	Only runs `IAlwaysRunResultFilters`.
Resource filters	Set `context.Result`	Resource-filter `*Executed` functions from earlier filters run with `context.Cancelled = true`. Runs `IAlwaysRunResultFilters` before executing the `IActionResult`.
Action filters	Set `context.Result`	Only bypasses action method execution. Action filters earlier in the pipeline run their `*Executed` methods with `context.Cancelled = true`, then result filters, result execution, and resource filters' `*Executed` methods all run as normal.
Page filters	Set `context.Result` in `OnPageHandlerSelected`	Only bypasses page handler execution. Page filters earlier in the pipeline run their `*Executed` methods with `context.Cancelled = true`, then result filters, result execution, and resource filters' `*Executed` methods all run as normal.
Exception filters	Set `context.Result` and `Exception.Handled = true`	All resource-filter `*Executed` functions run. Runs `IAlwaysRunResultFilters` before executing the `IActionResult`.
Result filters	Set `context.Cancelled = true`	Result filters earlier in the pipeline run their `*Executed` functions with `context.Cancelled = true`. All resource-filter `*Executed` functions run as normal.

The most interesting point here is that short-circuiting an action filter (or a page filter) doesn't short-circuit much of the pipeline at all. In fact, it only bypasses later action filters and the action method execution itself. By primarily building action filters, you can ensure that other filters, such as result filters that define the output format, run as usual, even when your action filters short-circuit.

The last thing I'd like to talk about in this chapter is how to use DI with your filters. You saw in chapter 10 that DI is integral to ASP.NET Core, and in the next section you'll see how to design your filters so that the framework can inject service dependencies into them for you.

13.4 Using dependency injection with filter attributes

In this section you'll learn how to inject services into your filters so you can take advantage of the simplicity of DI in your filters. You'll learn to use two helper filters to achieve this, `TypeFilterAttribute` and `ServiceFilterAttribute`, and you'll see how they can be used to simplify the action filter you defined in section 13.2.3.

The previous version of ASP.NET used filters, but they suffered from one problem in particular: it was hard to use services from them. This was a fundamental issue with implementing them as attributes that you decorate your actions with. C# attributes don't let you pass dependencies into their constructors (other than constant values), and they're created as singletons, so there's only a single instance for the lifetime of your app.

In ASP.NET Core, this limitation is still there in general, in that filters are typically created as attributes that you add to your controller classes, action methods, and Razor Pages. What happens if you need to access a transient or scoped service from inside the singleton attribute?

Listing 13.13 showed one way of doing this, using a pseudo-service locator pattern to reach into the DI container and pluck out `RecipeService` at runtime. This works but is generally frowned upon as a pattern, in favor of proper DI. How can you add DI to your filters?

The key is to split the filter into two. Instead of creating a class that's both an attribute and a filter, create a filter class that contains the functionality and an attribute that tells the framework when and where to use the filter.

Let's apply this to the action filter from listing 13.13. Previously I derived from `ActionFilterAttribute` and obtained an instance of `RecipeService` from the `context` passed to the method. In the following listing, I show two classes, `EnsureRecipeExistsFilter` and `EnsureRecipeExistsAttribute`. The filter class is responsible for the functionality and takes in `RecipeService` as a constructor dependency.

Listing 13.18 Using DI in a filter by not deriving from `Attribute`

```
public class EnsureRecipeExistsFilter : IActionFilter
{
    private readonly RecipeService _service;
    public EnsureRecipeExistsFilter(RecipeService service)
    {
        _service = service;
    }
```

Doesn't derive from an Attribute class

RecipeService is injected into the constructor.

```
    public void OnActionExecuting(ActionExecutingContext context)
    {
        var recipeId = (int) context.ActionArguments["id"];
        if (!_service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }

    public void OnActionExecuted(ActionExecutedContext context) { }
}

public class EnsureRecipeExistsAttribute : TypeFilterAttribute
{
    public EnsureRecipeExistsAttribute()
        : base(typeof(EnsureRecipeExistsFilter)) {}
}
```

The rest of the method remains the same.

You must implement the Executed action to satisfy the interface.

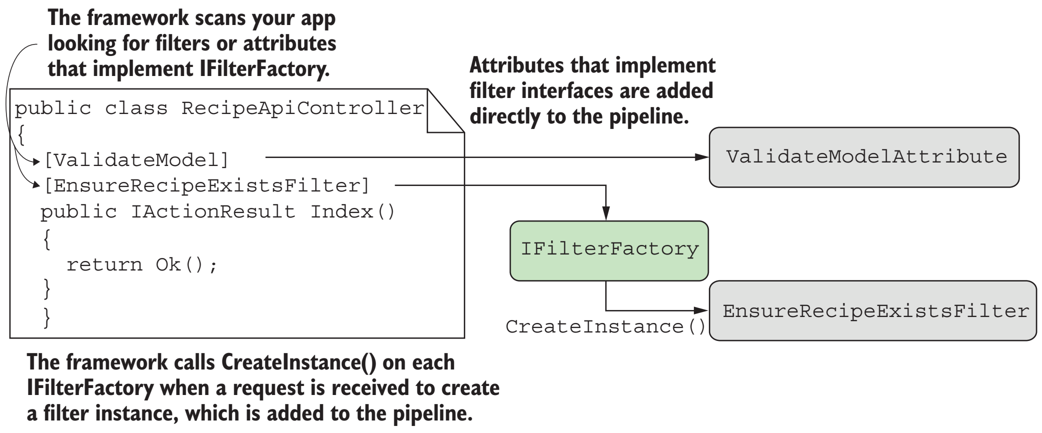
Passes the type EnsureRecipeExistsFilter as an argument to the base TypeFilter constructor

Derives from TypeFilter, which is used to fill dependencies using the DI container

`EnsureRecipeExistsFilter` is a valid filter; you could use it on its own by adding it as a global filter (as global filters don't need to be attributes). But you can't use it directly by decorating controller classes and action methods, as it's not an attribute. That's where `EnsureRecipeExistsAttribute` comes in.

You can decorate your methods with `EnsureRecipeExistsAttribute` instead. This attribute inherits from `TypeFilterAttribute` and passes the `Type` of filter to create as an argument to the base constructor. This attribute acts as a *factory* for `EnsureRecipe-ExistsFilter` by implementing `IFilterFactory`.

When ASP.NET Core initially loads your app, it scans your actions and controllers, looking for filters and filter factories. It uses these to form a filter pipeline for every action in your app, as shown in figure 13.8.


The framework scans your app looking for filters or attributes that implement IFilterFactory.
Attributes that implement filter interfaces are added directly to the pipeline.
ValidateModelAttribute
IFilterFactory
CreateInstance()
EnsureRecipeExistsFilter
The framework calls CreateInstance() on each IFilterFactory when a request is received to create a filter instance, which is added to the pipeline.

```
public class RecipeApiController
{
  [ValidateModel]
  [EnsureRecipeExistsFilter]
  public IActionResult Index()
  {
    return Ok();
  }
  }
```

Figure 13.8 The framework scans your app on startup to find both filters and attributes that implement `IFilterFactory`. At runtime, the framework calls `CreateInstance()` to get an instance of the filter.

When an action decorated with `EnsureRecipeExistsAttribute` is called, the framework calls `CreateInstance()` on the attribute. This creates a new instance of `EnsureRecipeExistsFilter` and uses the DI container to populate its dependencies (`RecipeService`).

By using this `IFilterFactory` approach, you get the best of both worlds: you can decorate your controllers and actions with attributes, and you can use DI in your filters. Out of the box, two similar classes provide this functionality, which have slightly different behaviors:

- `TypeFilterAttribute`—Loads all of the filter's dependencies from the DI container and uses them to create a new instance of the filter.
- `ServiceFilterAttribute`—Loads the filter *itself* from the DI container. The DI container takes care of the service lifetime and building the dependency graph. Unfortunately, you also have to explicitly register your filter with the DI container in `ConfigureServices` in `Startup`:

```
services.AddTransient<EnsureRecipeExistsFilter>();
```

Whether you choose to use `TypeFilterAttribute` or `ServiceFilterAttribute` is somewhat a matter of preference, and you can always implement a custom `IFilterFactory` if you need to. The key takeaway is that you can now use DI in your filters. If you don't need to use DI for a filter, then implement it as an attribute directly, for simplicity.

TIP I like to create my filters as a nested class of the attribute class when using this pattern. This keeps all the code nicely contained in a single file and indicates the relationship between the classes.

That brings us to the end of this chapter on the filter pipeline. Filters are a somewhat advanced topic, in that they aren't strictly necessary for building basic apps, but I find them extremely useful for ensuring my controller and action methods are simple and easy to understand.

In the next chapter, we'll take our first look at securing your app. We'll discuss the difference between authentication and authorization, the concept of identity in ASP.NET Core, and how you can use the ASP.NET Core Identity system to let users register and log in to your app.

Summary

- The filter pipeline executes as part of the MVC or Razor Pages execution. It consists of authorization filters, resource filters, action filters, page filters, exception filters, and result filters. Each filter type is grouped into a *stage* and can be used to achieve effects specific to that stage.
- Resource, action, and result filters run twice in the pipeline: an `*Executing` method on the way in and an `*Executed` method on the way out. Page filters

run three times: after page handler selection, and before and after page handler execution.

- Authorization and exception filters only run once as part of the pipeline; they don't run after a response has been generated.
- Each type of filter has both a sync and an async version. For example, resource filters can implement either the `IResourceFilter` interface or the `IAsyncResourceFilter` interface. You should use the synchronous interface unless your filter needs to use asynchronous method calls.
- You can add filters globally, at the controller level, at the Razor Page level, or at the action level. This is called the *scope* of the filter. Which scope you should choose depends on how broadly you want to apply the filter.
- Within a given stage, global-scoped filters run first, then controller-scoped, and finally, action-scoped. You can also override the default order by implementing the `IOrderedFilter` interface. Filters will run from lowest to highest `Order` and use scope to break ties.
- Authorization filters run first in the pipeline and control access to APIs. ASP.NET Core includes an `[Authorization]` attribute that you can apply to action methods so that only logged-in users can execute the action.
- Resource filters run after authorization filters, and again after an `IActionResult` has been executed. They can be used to short-circuit the pipeline, so that an action method is never executed. They can also be used to customize the model-binding process for an action method.
- Action filters run after model binding has occurred, just before an action method executes. They also run after the action method has executed. They can be used to extract common code out of an action method to prevent duplication. They don't execute for Razor Pages, only for MVC controllers.
- The `ControllerBase` base class also implements `IActionFilter` and `IAsyncActionFilter`. They run at the start and end of the action filter pipeline, regardless of the ordering or scope of other action filters. They can be used to create action filters that are specific to one controller.
- Page filters run three times: after page handler selection, after model binding, and after the page handler method executes. You can use page filters for similar purposes as action filters. Page filters only execute for Razor Pages; they don't run for MVC controllers.
- Razor Page `PageModels` implement `IPageFilter` and `IAsyncPageFilter`, so they can be used to implement page-specific page filters. These are rarely used, as you can typically achieve similar results with simple private methods.
- Exception filters execute after action and page filters, when an action method or page handler has thrown an exception. They can be used to provide custom error handling specific to the action executed.

- Generally, you should handle exceptions at the middleware level, but you can use exception filters to customize how you handle exceptions for specific actions, controllers, or Razor Pages.
- Result filters run just before and after an `IActionResult` is executed. You can use them to control how the action result is executed, or to completely change the action result that will be executed.
- Result filters aren't executed when you short-circuit the pipeline using authorization, resource, or exception filters. You can ensure result filters also run for these short-circuit cases by implementing a result filter as `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter`.
- You can use `ServiceFilterAttribute` and `TypeFilterAttribute` to allow dependency injection in your custom filters. `ServiceFilterAttribute` requires that you register your filter and all its dependencies with the DI container, whereas `TypeFilterAttribute` only requires that the filter's dependencies have been registered.